"An Introduction to Aspect Oriented Programming in *e*"

David Robinson

david.robinson@verilab.com

Rel_1.0: 21-JUNE-2006

This white paper is based on the forthcoming book “What's so wrong with patching anyway? A pragmatic guide to using aspects and *e*”.

Please contact david.robinson@verilab.com for more information.

Preface

"But when our hypothetical Blub programmer looks in the other direction, up the power continuum, he doesn't realize he's looking up. What he sees are merely weird languages. He probably considers them about equivalent in power to Blub, but with all this other hairy stuff thrown in as well. Blub is good enough for him, because he thinks in Blub"

Paul Graham[1]

This paper is taken from a forthcoming book about Aspect Oriented Programming. Not just the academic stuff that you'll read about elsewhere, although that's useful, but the more pragmatic side of it as well. It's about using AOP in ways that will make your code easier to write, easier to use, easier to reuse, and in a way that will help you meet your project schedule.

It gives real examples of AOP in action, and throws in some guidelines on how to organise your code so that you can actually find things again. Along the way it describes what an aspect really is and why OOP isn't the panacea the Blub programmers in our midst make out. It might even give you some ideas on how to avoid problems in OOP.

It's aimed at people who want to learn the truth about Aspect Oriented Programming (AOP). In particular, you'll find this book useful if:

- you're an expert in OOP but wary about AOP because of what you've heard. It will show you that not only does AOP improve OOP and

[1] http://www.paulgraham.com/avg.html

make software development easier, but it can do both of these without bringing an end to life as we know it

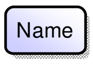
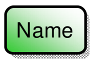
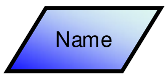
- you're someone who uses AOP in their daily lives but are not sure what it is you are using. Don't worry; no one else seems to know either. This book will explain what it is, show you when you should use it and when you shouldn't, and give some useful tips about how to deal with it all
- you're someone who uses AOP in their job and are sure what it is you are using. Please get in touch - I'd love to hear more. Hopefully, this book will give you some fresh ideas to try out, or at least be entertaining

Typographical conventions

"This book was typeset without the aid of troff, eqn, pic, tbl, yuc, ick, or any other idiotic Unix acronym."

The UNIX Haters Handbook

This book was written entirely, and with no problems whatsoever, using Word and PowerPoint, partly to annoy those who keep telling me it can't be done. I've used `courier` to represent code, and *`italicised courier`* to show comments in the code. Commands, such as Specman or Unix commands, are shown using **`bold courier`**, and program output is shown using **`indented courier in a slightly smaller font size`**. Filenames are shown in **bold dark grey**.

Symbol	Represents	Comments
	A concern	See page 11.
	A class	See page 19.
	A conceptual aspect	This represents a conceptual aspect, and has no reference to the actual implementation of the aspect. For instance, the code in this aspect could be spread across four different files, but a single symbol would be used to represent it. I use the UML's package symbol if I need to discuss aspects in terms of implementation structure.

About Verilab

Verilab is an international silicon engineering consultancy, focusing on the verification and design of advanced digital systems. Verilab has helped some of the most advanced VLSI engineering teams in the world improve their capabilities even further. Working at all levels, from verification planning and testbench architecture, through module and system level verification, to entire flow re-engineering, Verilab's expert consultants enable their client teams to achieve tangible, profit-enhancing reductions in design cycle times, increased confidence in the quality of the final device and lasting ruggedization and re-usability in design and verification infrastructure.

1 Introduction

A language that doesn't affect the way you think about programming, is not worth knowing.

Stephen C. Johnson

If your only exposure to programming technology has been through SystemVerilog or SystemC, you might be forgiven for thinking that object orientation is a brand new idea that's at the cutting edge of software design. After all, the marketing for these languages makes a big deal out of the fact that they are object oriented. While object orientation is a great thing to have in your language, it can hardly be classed as cutting edge. C++, which was released in 1983, drew inspiration from Simula 67 which was about in 1967.

Because object orientation has been around for a while, people have had time to get to know what works and what doesn't. As you'll see from the quotes I've used in the book, people have had enough problems with it to try some new things. Aspect orientation is one of those new things. Although aspect orientation is still classed as new, the first accepted publication using the name was in 1997[2] [1] and the concepts have been around since at least 1978 [2].

[2] *e* was around in 1992, which you'll notice is 5 years before AOP was officially developed. The creators of *e* drew their inspiration from many novel programming techniques which were being developed at the time, such as Subject Oriented Programming and Adaptive Programming [10]. So did the developers of AOP [8]. AOP went on to become the most popular of the various approaches, hence *e* being labelled as Aspect Oriented.

In its originally published form [1], aspect orientation is simply an additional way of structuring and encapsulating software. It's normally used in conjunction with object orientation to let you encapsulate a problem's aspects as well as its structure. I'll explain what an aspect is later, but for now, this simply makes it easier to write code. It isn't very exciting though, unless you are seriously into writing reusable software. This is what most of the mainstream literature focuses on.

Although not part of the original motivation, there is another use of aspect orientation that makes it much more interesting. It's the ability to cleanly add new features to the code without having to intrusively modify the code base. It doesn't sound like much, but it can be a life saver when deadlines loom. Depending on what's going through your mind when you're doing it, it can either be a shameful act (patching) or a wonderfully modern approach to building flexible designs (modular use-case driven development [2]). The boundaries between the shame of patching and the glory of modular use cases are pretty vague, and probably not worth getting too concerned about.

So what is an aspect? That's a good question, and one with a rather hard to explain answer.

2 What are aspects - part I?

"Aspect Orientation is primarily a mindset"

Markus Völter [4]

In one respect an aspect is like a class – it can be quite difficult to describe exactly what it is. Sure, you can tell people how a class is a grouping of related variables and functions, and that it's a template for an object which has state, behaviour and identity. You can then go on to talk about inheritance, encapsulation and polymorphism, but to get across the *essence* of a class, you tend to end up hand waving at a whiteboard using endless examples that are perfectly obvious to those who know about classes, but a mystery to those who don't.

Aspects make classes look easy. The problem is that an aspect is more abstract and less concrete than a class, so you need to do more hand waving. To start with, at least in *e*, there is no aspect keyword, so straight away you're trying to describe something that doesn't really exist, at least not in a "look, it's the bit after the word 'aspect'" kind of way.

I'm going to use the following example to introduce you to the concepts involved in aspect orientation. It's a very simple example, and has absolutely nothing to do with *e*, testbenches or object oriented programming. Figure 1 shows the topology of a basic design. It is composed of components, such as a CPU, a DMA, and a local memory controller (LMU), and these are connected together by a bus.

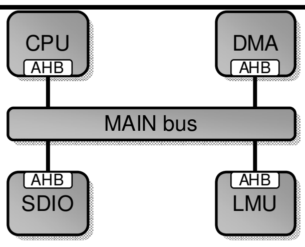


Figure 1: The basic design

Although this is a trivial example, it already shows many features relevant to aspect oriented programming, even if it isn't coded using aspects. Let's start with a loose definition. A *concern* is something in the design that you are concerned about. It is something you are interested in, and would probably like to encapsulate to make it reusable or to make it easy to review or replace. The most obvious concerns in the example are the components. Each component is a concern because you would typically want to be able to isolate it from the others so you can reuse it, assign it to one person to design, have it all together in one place for review and so on. In Verilog, each component would be encapsulated as a module. In VHDL it would be an entity and in SystemC it would be a class.

The bus is also a concern. Why? Because it is something you want to encapsulate and deal with on its own. However, there is something special about this concern; something that makes it different from the individual component concerns. The difference is that it doesn't exist on its own. Instead, it physically cuts across (or interacts) with other concerns. Because of this, it's known as a *cross cutting concern*. In this example it cuts across four other concerns – the CPU, SDIO, DMA and LMU (Figure 2). In order to work with the bus, AHB interface code has to be physically added to these components.

In Figure 2, the outer boxes represent the container (module, entity or class) used to encapsulate the component concerns. The inner boxes represent the code that implements the appropriate concerns. You'll see that while the code for the component functionality is fully contained within the container, the code for the bus interface concern is in all four containers. It cuts across the component concerns.

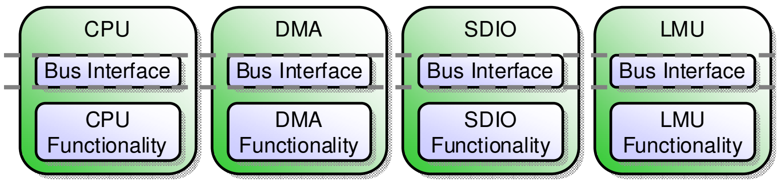


Figure 2: The bus cross cutting concern

They don't seem like much, but cross cutting concerns are at the heart of AOP and are the reason that the entire methodology was created. Cross cutting concerns cannot be encapsulated just using object orientation, and that causes problems.

So if cross cutting concerns cause problems, is there anything we can do about them? Yes, there is, and that's where aspect oriented languages come in. They provide language constructs that let you encapsulate cross cutting concerns. Non-cross cutting concerns are coded using OOP techniques as before, and new language constructs are used to code the cross-cutting concerns and integrate them with the non-cross cutting concerns. That's right, "*non-cross cutting concerns are coded using OOP techniques as before*". So straight away all of the OOP experts can relax. OOP's not being thrown away. It's simply being augmented to make it work a bit better.

For now, just assume that an aspect and cross cutting concern are the same thing. An aspect is actually more than this, but that can wait until later (until page 40 to be exact). I'd like to finish off the example to show that you can use aspects without an aspect oriented programming language[3]. If this example design is to be turned into silicon, then it needs to have a clock tree and scan chains added. These are also cross cutting concerns (Figure 3).

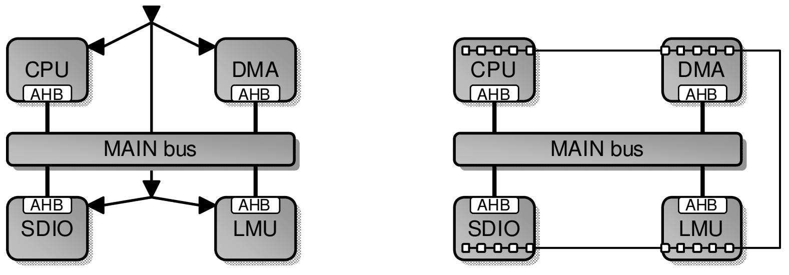


Figure 3: The clock tree and scan chain concerns cutting across the component concerns

As they are cross cutting concerns, the code that implements the components has to be physically modified to contain the code required to implement the clock tree and scan chains.

[3] My favourite definition of AOP is that aspects are a state of mind. Why? Because aspects are really just a way of thinking about the functionality in your code. You don't actually need language support to think in aspects, which *e* demonstrates admirably by not having an aspect keyword. Aspect Oriented languages do have some language features to help implement aspects, but you can do most of it with OOP design patterns if you are *really* keen [4].

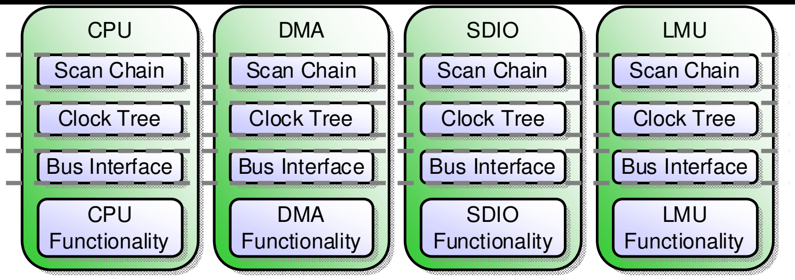


Figure 4: The code structure with all concerns

However, if you open up the source code for any component in an ASIC design, you'll rarely see any clock trees or scan chains. Why? Because these are automatically dealt with by software tools. Information about the clock and scan requirements are encapsulated elsewhere and the required code is automatically woven in with the components' code. This is aspect oriented programming, which isn't directly supported by Verilog, VHDL or SystemC.

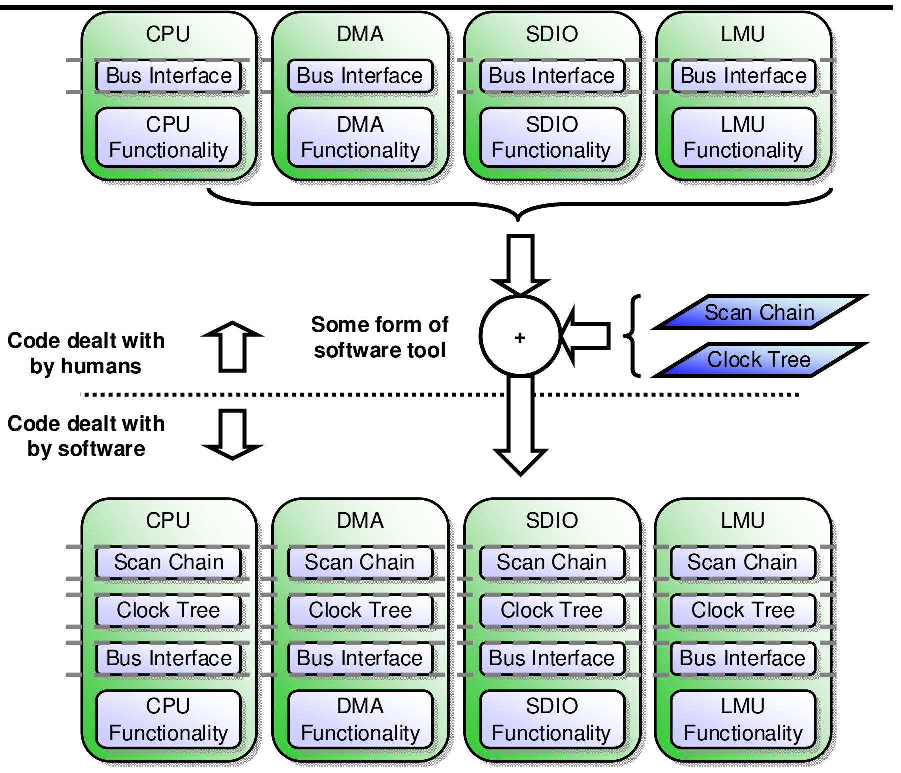


Figure 5: Handling aspects in an HDL

3 Why do I need aspects? What's wrong with cross cutting concerns?

"If you could do Java over again, what would you change?"

"I'd leave out classes"

James Gosling, inventor of Java

Let's return to our example to show why cross cutting concerns are a problem. Rather than look at the topology of the design, let's look at the code that the designers have to deal with. I'm including the clock tree and scan chain concerns in the components' code because I'm trying to show the problems you'll have if you don't use aspects.

Code that has unencapsulated cross cutting concerns, like in Figure 6, has two undesired features. *Code scattering* is where the code for a particular concern appears in multiple places in the design, and *code tangling* is where the code for multiple concerns is tangled together, forcing an object to deal with multiple concerns simultaneously.

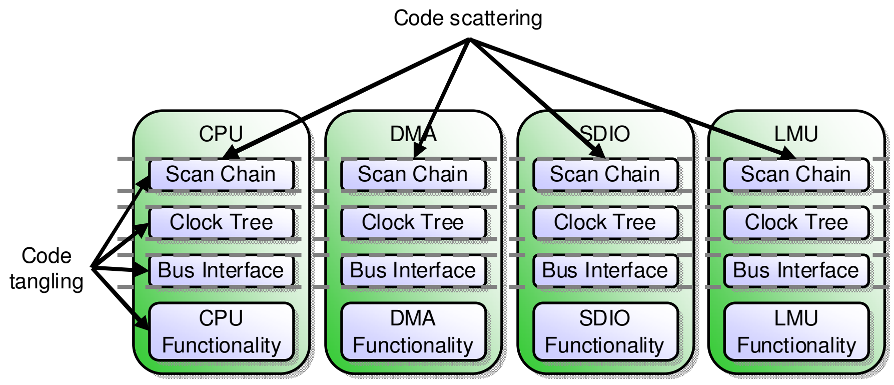


Figure 6: Showing the code scattering and code tangling in Figure 4

The code for the bus interface is scattered between all of the components, as is the code for the clock tree and the scan chains. If you just concentrate on the CPU, you'll see that apart from the code required to implement the CPU functionality, it also contains the code required to implement the bus interface, the clock tree and the scan chains.

These two features may appear innocuous, but they can cause a wide variety of problems:

- Code cannot be reused because it contains part of a cross cutting concern that doesn't exist in the new target:
  - none of the components in the example can be used in an FGPA design because of the scan chains and clock trees
  - none of the components can be used on a non AHB bus
- It can be hard to maintain the code:

- any changes to the bus interface will require code updates in multiple places
- the tangled code obscures the core functionality of a component, so it's hard to focus on the essential code

- It can be easy to introduce bugs:
  - it's hard to ensure that all scattered code is written consistently
  - it's hard to ensure that all instances of scattered code are updated if a new feature is added
  - without an owner for each cross cutting concern, it's possible that it isn't implemented in all of the places it should be
  - the designer has to deal with the cross cutting concerns while trying to write the functionality of the non-cross cutting concern

Aspects let you encapsulate cross cutting concerns, so these problems can be made to go away.

4 Surely OOP doesn't have any problems?

"Most OOP ideas were alive and well 20 to 25 years ago. Why didn't they solve the software problem then"

Stephen C. Johnson

I mentioned earlier that cross cutting concerns are a problem. You might be thinking that they aren't a problem for you because you use OOP to write your testbench, and that's state of the art, isn't it?

Well no, it's not. OOP is old technology, and AOP grew out of a realisation that OOP is not as perfect as people seem to think. OOP has been used on many projects now, and many practitioners started to realise quite early on that something was wrong. Even when a problem was mapped to classes, programmers still had to concentrate on many different things at once, software maintenance had not become trivial, and simple-and-widespread class reuse remained elusive [9].

The root cause of these problems is that OOP doesn't allow you to decompose a problem into *all* of its concerns. You can only encapsulate some of the concerns. It doesn't easily allow you to deal with cross cutting concerns.

The only structure supported by OOP for encapsulation is the class[4]. When mapping a problem to classes, you have to decide which concerns are the

[4] *e* doesn't have a *class* type. Instead, it has structs and units which from a software point of view are classes. There are differences between them, such as when they can be created, whether or not they can control objections, if they can have hdl_paths, etc, but these are unimportant for this book, so I will refer to both units and structs as classes

most important, and turn them into classes. This gives you the structure of your testbench code. The concerns which you pick to encapsulate in classes are called _dominant concerns_, and these should be independent of each other. Therefore, you could pick CPU and DMA to be dominant concerns because these can be fully encapsulated without knowledge of each other, but you shouldn't pick CPU and monitoring because these are cross cutting (Figure 7). There is no way to encapsulate these in OOP so that they don't have any knowledge of each other. The CPU dominant concern must contain information about monitoring, and the monitoring concern must contain information about CPU. This is poor encapsulation.

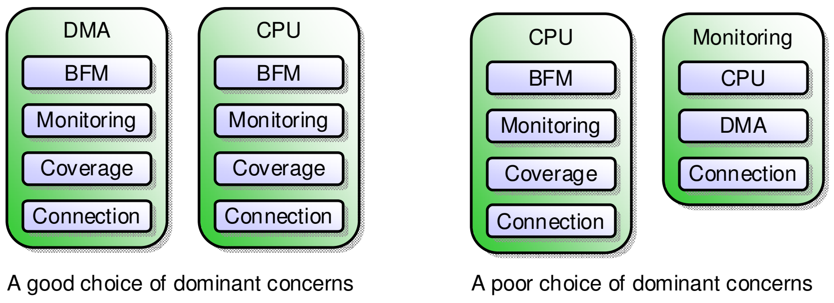


Figure 7: Good and bad choices of dominant concerns

The remaining cross cutting concerns have to fend for themselves. There is no way to fully encapsulate them in OOP. You could try to create a class to encapsulate a cross cutting concern, such as functional coverage, but you won't succeed. While you could encapsulate most of it, any concerns that it cuts across still need to instantiate this new class and interact with it in the correct manner. This means there is still some unencapsulated code for the cross cutting concern.

In this example the other classes still need to know that there is functional coverage, and they need to instantiate the functional coverage class and to interact with it. The code required to do this is unencapsulated code. It is nothing to do with the dominant concerns, yet it appears in their code. We couldn't take the dominant concern classes to a new project without taking the functional coverage class with them, and there isn't one place we can go to review all of the functional coverage in the testbench.

So if OOP will only let you encapsulate dominant concerns, what should you make the dominant concerns? Perhaps if you make the correct choice then all of the problems go away?

Let's look at the testbench code required to verify the SDIO component from the example. The SDIO is a serial transmitter and receiver that supports three different protocols - SD-N (1 bit), SD-W (4 bits) and SPI. I'll ignore the clock tree and scan chains because they don't exist during functional verification. I'll also ignore the bus interface, because I want to concentrate on the protocols themselves. So what are the dominant concerns?

Dominant concerns should be orthogonal and you would not normally consider the entire testbench to be a concern. This rules out creating one large class that does *everything* to do with SDIO.

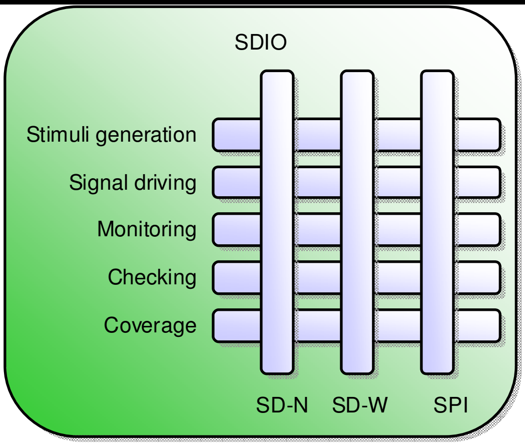


Figure 8: Using the SDIO testbench as the dominant concern

You could pick the protocols themselves as the thing you want to encapsulate, and design a single class that encapsulated everything to do with verifying, say, the SD-N protocol.

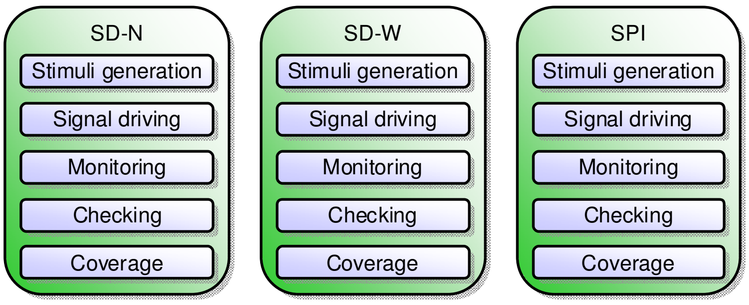


Figure 9: Structuring your code when protocol is the dominant concern

But there's a problem with this. For one class to encapsulate everything about the SD-N protocol, then that class would have to generate the stimuli, drive the signals, monitor the signals, check the behaviour, and record the functional coverage, and therein lies the problem - programmers intuition, common sense, the entrails and the eRM tell you that you should have individual classes for these. It just feels like a more natural way to do it, probably because each of these tasks are orthogonal and appear as valid candidates for dominant concerns in their own right.

If you code this way, then you find that the SD-N protocol code is now smeared across several classes (Figure 10). There is some in the BFM, there is some in the monitor and there is some in the checker. The problem now is that the SD-N protocol is no longer encapsulated.

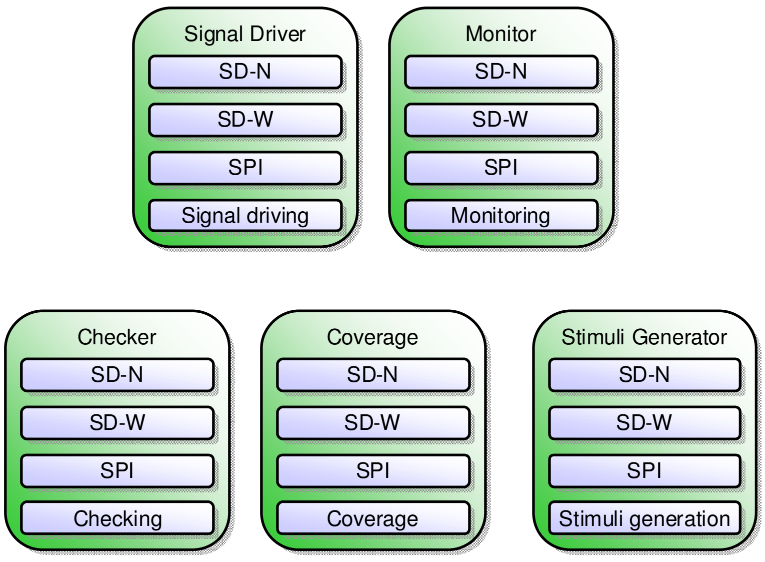


Figure 10: Structuring your code when testbench components are the dominant concern

In reality, you would compromise with this approach and create stimuli generators, monitors, signal drivers, checkers and coverage objects for each of the protocols. This will allow you to do some encapsulation of the protocols and of the testbench components. However, this results in poor encapsulation of everything (Figure 11). Not only are the protocols unencapsulated, but so are the testbench components. There is no longer one class that encapsulates everything about, say, the monitoring[5].

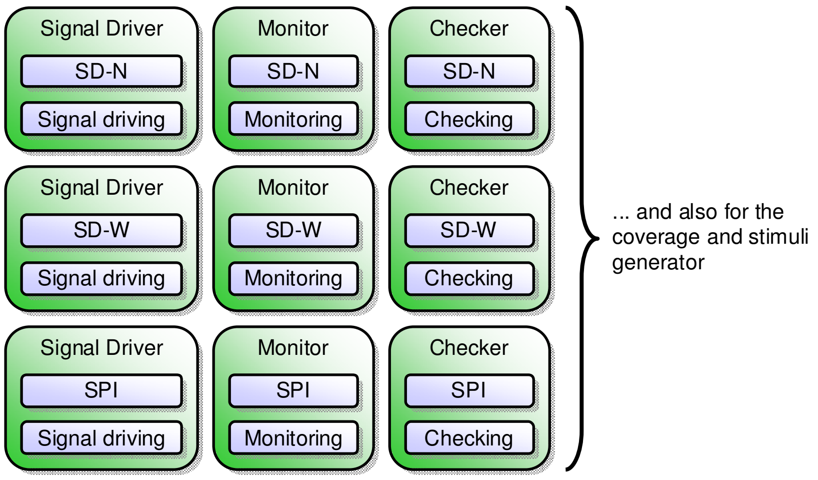


Figure 11: A compromise for when testbench components are the dominant concern

This seems like an acceptable solution, but let's look at what happens if you start taking other concerns into consideration. For instance, the way you

[5] You may think that there's no need to encapsulate all monitors in the testbench, but that's probably because it's never really been plausible to do this without AOP. Now it is a plausible thing to do, you might see some benefit to it. For instance, all monitors can be easily isolated and reviewed to check performance or logging

connect your testbench to the DUT can be a concern, and we haven't considered that at all. Let's say in your project you want to ensure that all signal maps are implemented using ports, and that all signals are accessed using methods to deal with some programmable polarities your design has. Your signal drivers and monitors all contain signal connection code, and you have one each of these for each protocol. In the above example, your signal connect code would be spread amongst six different classes. Perhaps if you throw away the structure we already have and make signal connection a dominant concern then your encapsulation will improve?

Figure 12 shows a class structure that would provide ideal encapsulation of your signal connections. This really isn't any better than before. Again, a more pragmatic approach would normally be taken and multiple signal connect classes would be created - one per protocol or one per monitor-and-checker per protocol.

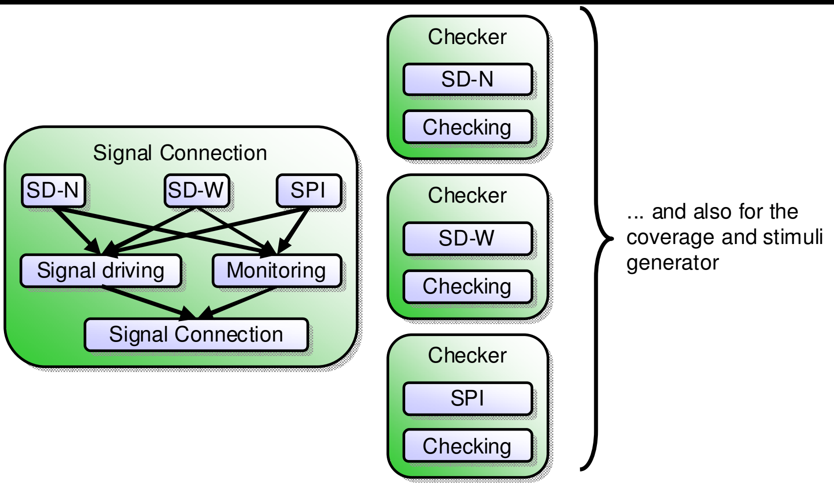


Figure 12: The ideal encapsulation of signal connections

I think it's fair to say that there is no "good" set of dominant concerns in this example that allows everything to be fully encapsulated. No matter what concerns you pick as your dominant concerns, the others become badly encapsulated. This is known as *the tyranny of dominant decomposition* [7].

Figure 13 shows the signal driver and the monitor classes for the best, and most common, compromise I spoke about earlier. Notice that even when you make the best decisions you can about what should be the dominant concerns, and how you structure your code, scattering and tangling still occur.

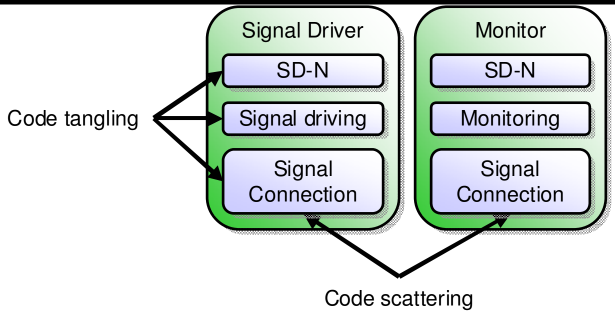


Figure 13: The encapsulation of the SD-N protocol

Many OOP practitioners are unaware of these problems because they are so used to working around them.

5 Why does AOP help?

"We can send a message around the world seven times in one second, but it still takes 20 years for that message to go through a quarter inch of human skull."

Stephen C. Johnson

This book goes into a lot of detail about how AOP can be used to solve the cross cutting concern problem (and some others as well), but it is probably a good idea to explain here *why* AOP solves the problems that OOP has.

To recap, the problem with OOP is that it can only properly encapsulate one concern in a class. All other concerns have to be scattered throughout the testbench and tangled with the other code. The class may contain bits of other concerns, but only a dominant concern will exist *fully* within it.

There are only two things that you need to add to an OOP language to turn it into an AOP language. You need the ability to add new members[6] to a class from files other than the one that declared the class, and you need the ability to add functionality to existing methods. These are known as *introduction* and *advice* respectively.

And that's it! That's all the extra stuff that an AOP language needs to support aspect orientation. That doesn't seem like much to base a new programming paradigm on, so why does it help? Let's take introductions first. In most (if not all) OOP languages, a class or class header has to be

[6] A *member*, or class member, is a property or a method in a class. It's used to describe something that belongs to a class.

defined in a single file. All of the properties and methods that a class needs must be declared in one file, and that leads to code tangling. If you need a property or a method to support a cross cutting concern, it has to be hardwired into the same file that declares all of the properties and methods needed by the dominant concern. Of course, this also leads to code scattering, because you'll need to put the same methods and properties into the definitions of all of the other classes that interact with the cross cutting concern.

However, if you can declare the class and then add new properties and members from other files, you can physically separate the cross cutting concerns from the dominant concern.

Well, almost. You can physically separate the properties and methods needed for the various concerns, but you still have to deal with the *use* of these properties and methods. That's where advice comes in. Advice lets you alter the functionality of an existing class by adding new functionality at *join points*, which are simply well defined points in the code. Method calls are the most common joint point, but *e* supports some others, such as coverage groups, coverage items and events[7]. For method advice (I'll talk about the others later), you can say "execute advice A before method M", "execute advice A after method M", or "execute advice A instead of method M".

[7] Defining these as join points stretches the commonly understood definition a bit because coverage groups, coverage items and events aren't part of the program's flow of execution. However, advice is really a behaviour modifier, and you can statically alter the behaviour of coverage groups, coverage items and events in *e*.

Why does this help? Well, you can now build up the behaviour of a method from several places. In the class definition which deals with the dominant concern, you can just deal with the dominant concern. In the files that deal with the cross cutting concerns, you can now hook in the necessary behaviour, assuming you have a join point at the right place[8].

So AOP is simply this: it's a way of structuring your code that lets you build a class up in individual pieces, or slices of functionality.

Figure 14 shows the scattering and tangling problem for a subset of the example in Figure 6. On the left are the concerns - the functional areas that the testbench has to deal with. In this example, the concerns we are interested in are the CPU and DMA core functionality along with the bus interfaces, the clock tree and the scan chains.

On the right are the classes that exist within the design. These are the classes you edit and they are the classes that get executed. As the only place code can be encapsulated is in these classes, code for the clock tree appears scattered in two classes, and in each class, it is tangled with some other code (scan chain, bus interfaces, etc). This is shown by the concerns embedded inside the classes. If you were to open up the file for one of these classes, you would see scan chain code, clock tree code, bus interface code and core functionality code.

[8] This means you'll want to write your code using as many methods as you can. This is now generally accepted [13] as good practice even without AOP

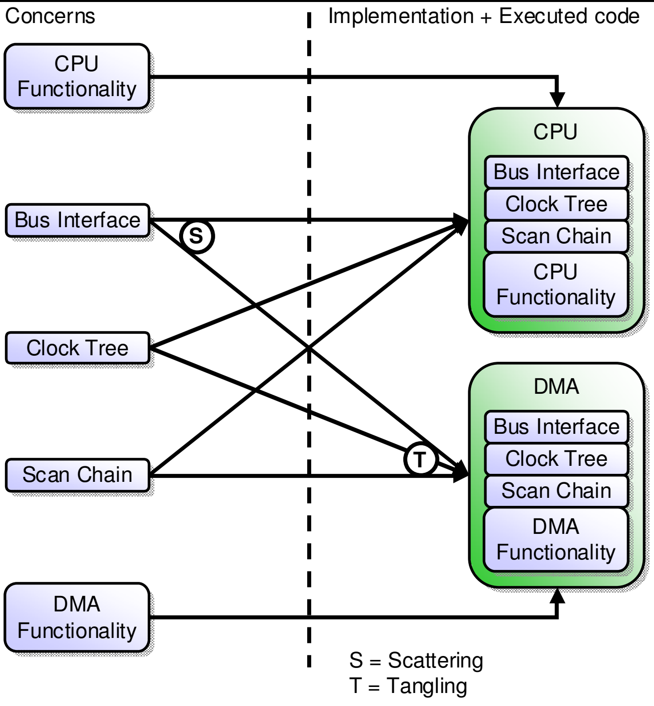


Figure 14: The scattering and tangling problem

Figure 15 shows how aspects remove the scattering and tangling problems. In the middle of the diagram are aspects and classes. Each class contains the class definition and the core functionality encapsulated by the class, but nothing else. In particular, they contain nothing to do with clock trees, bus interfaces or scan chains. Each aspect extends these classes to add the advice and introductions necessary to deal with a particular cross cutting concern, such as clock tree or scan chain.

Note that there is no scattering or tangling of code *within* the aspects. If we want to see *everything* to do with clock trees, we open up the clock tree aspect.

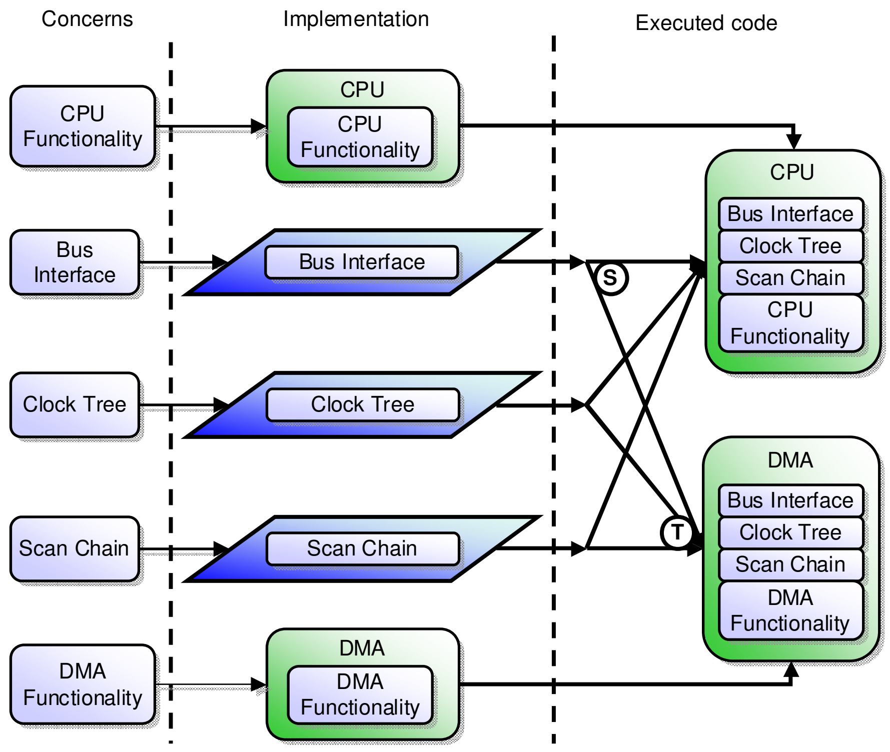


Figure 15: How aspects remove the scattering and tangling problem

Of course, this changes after the weaving stage[9], and the code is as tangled and scattered as before. You can see this on the right hand side of the diagram. However, this is an automatic process as part of compile step, so you never have to deal with this code directly. In fact, you don't even get to see it. All you deal with are the classes and aspects in the middle of the diagram.

[9] The *weaver* is the tool that integrates the aspects with the objects. Conceptually, it acts like a pre-processor that takes the class, physically adds all introductions to the class definition, and inserts the advice code into the join points. In reality, the Specman tool that processes and executes *e* does not have a pre-processing weaver because it can compile most of the *e* code and still apply extensions to it at run time by loading an interpreted *e* file [10]. This is a technical issue that is fairly unimportant. It helps to imagine it has a weaver. It is a good lie.

6 Theory vs. real life - what else is AOP good for?

"When academic ideals meet business realities, business realities usually win"

Ethan M. Rasiel

The previous section introduced AOP and explained why it was gaining in popularity in the software world. By allowing us to encapsulate cross cutting concerns, we can avoid the code scattering and code tangling problems. At this point it is probably worth noting that there are two types of verification engineer. The first, engineers with an interest in software, are either agreeing wholeheartedly with the above discussion about the problems with OOP, or claiming that I'm a blasphemous heretic. It's probably best that the latter stop reading now, because it's going to get a lot worse. The second group, hardware engineers struggling to write testbenches, are probably going "*huh?*" or "*so what?*". This section will try to address this group.

It is important to realise that many verification engineers today are just temporarily rebranded hardware engineers whose turn it is to do some verification. They haven't been trained to write software, they haven't been shown what can be achieved, many are simply not interested in making a proper job of it, and those who try frequently run into schedule problems because there is never enough time allocated to the task. After all, the testbench is not part of the product! Mainstream testbench design is unfortunately about solving problems *now*, without any care about how hard you may have made your life tomorrow.

Simply put, many testbenches are being written by people who are effectively unskilled in the job. Just as we wouldn't expect a software engineer to design a multi-clock domain ASIC device, why should we expect a hardware engineer to create a medium sized software program?

As a result of this, many testbenches are being written using poor naming conventions, insufficient commenting, poor object composition, and overuse and misuse of inheritance. They rely far too heavily on public member variables, contain a plethora of hardwired values and have poor abstraction and encapsulation. Many testbenches simply cannot be kept alive until the end of the project, never mind maintained and reused for derivative and next generation designs. With such an array of basic problems in testbenches, is it any wonder that discussions about cross cutting concerns are likely to fall on deaf and uncomprehending ears?

Perhaps the biggest advantage AOP has in testbench design then is not the ability to solve some high level problems, that are frankly pretty minor compared to the other problems in today's code, but its ability to help verification engineers to solve complex problems *now*. Yes, I'm talking about using AOP to patch code. Now, once the software puritans reading have stopped choking (and declaring me a blasphemous heretic), we can take time to look at what this actually means. You see, OOP also has some other problems that while not as conceptual as cross cutting concerns, are practical problems because they can cause you to miss your deadlines. Surely anything that can do this should warrant careful consideration?

If there is one feature that makes people distrust AOP, then it's its ability to patch code. Being able to change the functionality of a class without directly modifying the class source file seems to be abhorrent to some. "You can't read the code and know the functionality" they cry, neatly ignoring the fact that OOP has an *even worse* version of the same problem. When you have a pointer to a base class that has virtual methods, only a *runtime* analysis of the program will tell you what method actually gets called. At least with AOP, only a compile time analysis is needed[10]. As you will see in this book, patching code is simply a pragmatic programming technique that allows testbench designers to concentrate on finding DUT bugs, and not waste time trying to craft the perfect testbench.

So why is patching so useful? Well, imagine that you cannot patch code - that all changes made to a class must be made to its source file, or through inheritance and polymorphism. Also imagine that the class you are working with, a BFM for a packet based serial transmitter, has a bug in the `transmit()` method.

There are two options open to you to fix this bug. You can modify the original source code of the `transmit()` method or you can use inheritance to create a new subtype containing your bug fix. Modifying the source code is not always possible:

[10] If you use conditional AOP then you also have to do a runtime analysis, but all the information you need to work out what will get called is stored in the class's determinant fields, so it's not really an issue. Yes, I know I haven't defined conditional AOP yet, but the index is your friend.

- It may be commercial code that is encrypted, so you can't change it. You can ask the supplier for a bug fix, but that takes time
- The code may be referenced from a central repository, where you don't have write access. You can ask the original author to make the changes, but again, this takes time. Changing code in a repository has an impact on all other projects using it, and they will all have to be re-regressed. You could try to take a local copy, but your build environment may always reference the repository version
- The original code may be yours, and stored locally. While you are free to make the changes, you will prevent your testbench from compiling until it is done. If the bug is complex, this may take several days. In the meantime, you may need to run the testbench to recreate a DUT bug, which you now can't do[11]

So changing the original code may not be possible or desirable. The other option is to use inheritance to create an extended version of the class. The new class just contains a new version of the `transmit()` method with the bug fixed. There are two potential problems here. The first is that you now have a new class type, and the only way your new `transmit()` method will get called is if you create an object of your new type. If the BFM is not yours in the first place, then the chances are high that your code is not instantiating the BFM directly, which means you can't change the instantiation to use your new BFM class. To use your new class will require changes to other source code, which for the reasons stated above, may not be possible. In fact, if you

[11] You can use your revision control software to step back to an old version or to create a branch for your changes, but few people seem to do this.

can change the code that instantiates the BFM, you can probably change the BFM.

The second possible problem is quite technical. If the BFM is upcast anywhere in the code, i.e. it is accessed through a pointer which has the type of the base class, then the polymorphism will only work if the `transmit()` method in the base class is declared as virtual. If it is not virtual, the original `transmit()` method will continue to get called. It is very common for people not to declare methods as virtual, either because they are inexperienced and didn't know the impact of this, they are experienced but forgot to type `virtual`, or they are experienced and were deliberately closing off the class to further extensions.

This all means that in some quite common cases, you will not be able to fix a bug in a class using OOP. With AOP, the situation is very different. Because AOP allows you to change the functionality and structure of a class without having to modify the original code, all of the OOP problems go away. You don't need to have access to the original source code to replace a method with a new version, you don't have to create a new class type, and you don't have to remember to declare the original method in a special way. Encountering a bug like this while using an AOP language is not an issue.

The arguments above also hold if you want to add new functionality to a class, such as the ability to transmit a new protocol. An OOP language may force you to do some complex refactoring of the code, assuming you can get access to the code. With aspects, adding new functionality is trivial. Aspects therefore have another benefit. Because it is now easy to patch code, testbench writers only have to write the code that is absolutely necessary at that moment in time. They do not have to write functionality that *may* be useful at a future date [13].

Of course, wanton patching is poor technique, and can lead to very complex code. Once a bug has been fixed, or the new functionality added, the original source code should be updated with the changes. However, this can now be done at a suitable time of your choosing.

7 What are aspects - part II?

The definition I gave earlier of an aspect being a cross cutting concern is good, but it could be better. It seems like the most complex part of AOP is coming up with an exact definition for an aspect. This is not helped by the fact that the word 'aspect' is used for two different meanings. In OOP, a 'class' is the physical representation of a 'dominant concern'. In AOP, an 'aspect' is the physical representation of an 'aspect'. This isn't helpful.

Without an aspect keyword, *e* makes it hard to define the physical meaning of an aspect. To simplify things, let's assume that whatever we decide a conceptual 'aspect' to be, the physical 'aspect' is just the code representing an encapsulated version of it.

A common definition of an aspect is that it is a well modularised cross cutting concern [1]. This definition of an aspect describes what it conceptually is, but gives no aid in mapping this concept to actual code. Some other definitions offer more practical advice. Webopedia[12] uses the following definition:

> *"Aspects in aspect-oriented programming (AOP) package advice and pointcuts into functional units in much the same way that object-oriented programming uses classes to package fields and methods."*

In *e* this would be advice and introduction (pointcuts don't exist in *e*).

[12] http://www.webopedia.com/TERM/A/aspects.html

I find this all a bit intangible though, so I prefer to start with another definition of aspect, where it is a use-case or horizontal slice of functionality in the design [2]. The horizontal concept comes from assuming that your dominant concerns (classes) are vertical, and the aspects are stacked on top of them. Figure 4 shows what I mean by this. This helps me visualise the program, a testbench in our case, as a series of interwoven slices of functionality that can be added, removed and replaced without having to modify any code that isn't in the aspect.

But that isn't quite what I mean when I talk about aspects. I like to stretch the concept a bit further. The problem I have with an aspect being a horizontal slice of functionality is this - what happens when some code belongs to multiple slices of horizontal functionality? Do we not then end up with the same problems as before, that the code is scattered between different aspects? As an example, let's look at two use cases for a DMA controller:

- Configuring a transfer: The software writes to the DMA controller's configuration registers to set up a transfer
- Clearing an interrupt : The software writes to the DMA controller's interrupt register to clear an interrupt

Both of these require a slave bus interface, so which use case (or horizontal slice) does this belong to?

So I don't define aspects to be a 'horizontal slice' of functionality. I define them to be a 'slice' of functionality. I don't care which direction it's going in.

This might sound a bit picky, but it does have a big effect because it lets you assign code to multiple aspects, it lets you treat multiple aspects as one aspect, and it lets you treat a concern as an aspect, not just a cross cutting concern. It also lets you use the word aspect as it is defined in the dictionary - as a feature or facet.

A good way to visualise this is to imagine that your code is stored in a large database that you can query. You can ask the database to return just the code that deals with feature X, or where features X and Y interact, and it will do so. The query is a conceptual aspect, and the code returned is the physical aspect.

Let's go back to the example in Figure 1 (page 11). Imagine all of the code that defined this design, including the clock trees and scan chains, was stored in this database. What kind of queries could I make? What aspects are there to this design? Here are some examples:

Query	Returned code
Show me all the code for the CPU's core functionality	All the code for the CPU, but excluding the bus interface, scan chain and clock tree
Show me the code I've patched into the SDIO to temporarily fix the transmit bug	The code in this particular patch
Show me all the code where the CPU	The code where the bus interface

interacts with the AHB bus interface	interacts with the CPU
Show me all the code for the scan chains in the LMU and SDIO	The code where the scan chains interact with the LMU and the SDIO

The last two queries are examples of how an aspect can be made up of other aspects. The above queries deal with functional concerns, but I can define structure as a concern as well. I could say that I wanted to see all the code that dealt with CPU and monitoring, and what do I get? I get the CPU Monitor class!

So that's my definition of an *aspect* – it is a particular view of your code. The trick now is how to structure your code to make this happen. Without such a database, you'll have to organise your code so that you can isolate aspects. That means you'll have to manually define the aspects you are interested in, normally when you start writing the code. If the code for an aspect is scattered throughout the code base and cannot be isolated in some manner, then I don't consider it an aspect, just a mess. Remember that the original definition contained the words "well modularised".

There are several techniques for structuring your code, such as tagging the code (see the full book for details), storing all the code for an aspect in a single file, spreading the code across several specially named files (see the full book for details), storing all the code in a particular directory, etc. In this book, all files that combine to make up an aspect are referred to as *aspect files*.

So well done for making it this far. You now know all you need to know about the theory of AOP. The rest of the book just delves into technical details about how to use it in *e*, how to use it without making things worse for yourself, and what to use it for in verification environments.

What you do after the introduction will probably depend on how much *e* you already know. If you are new to *e* then it's probably worth reading chapter 2 in detail. If you know *e* already, then you can probably get away with just skimming chapter 2.

Chapters 3 and 4 give some ideas on what you can use aspects for in your verification environment, and how you should go about using them. Everyone should read these chapters.

Chapter 5 describes a script I wrote that I use to analyse *e* code. I use it a lot when I'm documenting my code, or if I'm working with someone else's code. It provides a quick, easy and free way of working out what you have and where it is. You will be able to download it for free from www.verilab.com.

Bibliography

AOP

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, ECOOP’97, June, 1997

[2] P. Ng, I. Jacobson, "Aspect-Oriented Software Development with Use Cases", Addison Wesley, January 2005

[3] Ramnivas Laddad, "I want my AOP!, Part 1", Java World, http://www.javaworld.com

[4] M. Völter, "Handling Cross-Cutting Concerns: AOP and beyond", Aug 2003, http://www.voelter.de/publications/index/detail723358759.html

[5] R. J. Walker, E. L.A. Baniassad and G. C. Murphy, “An Initial Assessment of Aspect-oriented Programming”, 21st International Conference on Software Engineering, May 1999

[6] B. Hayes, “The Post-OOP Paradigm”, American Scientist, Volume 91, Number 2, March–April, 2003

[7] H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns and The Hyperspace Approach.", Symposium on Software Architectures and Component Technology, 2000.

[8] C. Lopes, “Aspect Oriented Programming: A Historical Perspective (What’s in a Name?)”, Institute for Software Research Technical Report #UCI-ISR-02-5, 2002

AOP for Verification

[9] D. Robinson, J. Sprott and G. Allan, "Learn to do Verification with AOP? We've just learned OOP!", SNUG, 2004. http://www.verilab.com

[10] Y. Hollander, M. Morley and A. Noy, "The *e* Language: A Fresh Separation of Concerns", TOOLS Europe, 2001

[11] S. Regimbal, J. Lemire, Y. Savaria, G. Bois, E. Aboulhamid and A. Baron, "Aspect partitioning for Hardware Verification Reuse", International Workshop on System-on-Chip for Real-Time Applications, 2002

Software Techniques

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional, 1995

[13] M. Kircher, P. Jain, A. Corsaro, "XP + AOP = Better Software?", XP2002, 2002

[14] I. Jacobson, "The Case for Aspects", Software Development (http://www.sdmagazine.com/), 2003

Critiques of OOP

[15] A. Holub, "Why extends is evil", Java World, August 2003, http://www.javaworld.com/javaworld/jw-08-2003/jw-0801-toolbox.html

[16] D. Joiner, "Encapsulation, Inheritance and the Platypus effect", May, 2000, http://www.advogato.org/article/83.html

[17] J. K. Ousterhout, "Scripting: Higher-Level Programming for the 21st Century", IEEE Computer, March 1998, pp. 23-30. http://home.pacbell.net/ouster/scripting.html (Section 6: The role of objects)

[18] S. Johnson, “Objecting to Objects”, USENIX Conference, 1994

Index